# Fundamentos de Estructuras de Datos y Algoritmos

**Tarea 4**

Danilo Bonometti

## I. Introducción

En este trabajo se implementó un modelo simplificado de una red social. La red está representada por un grafo no dirigido donde los usuarios comparten una relación bidireccional. Los usuarios corresponden a los vértices del grafo y la relación entre ellos corresponde a las aristas entre cada vértice. De las implementaciones requeridas para trabajar con el modelo, podremos observar que la complejidad de redes como estas escala de manera tal que se dificulta de gran forma el estudio de sus características. En particular podremos observar como la búsqueda de comunidades en la red (representadas por subgrafos completos o *cliques*) tiene una complejidad exponencial con respecto a la cantidad de nodos en esta. Existen distintos algoritmos para enfrentar este problema. El más famoso y reconocido siendo el de Bron-Kerbosch [1] que usa una estrategia de retroceso recursivo para encontrar todos los cliques de forma óptima en términos de cantidad de cliques. Otros algoritmos expresan el problema de encontrar los cliques maximales en términos del problema SAT [2]. También hay algunos algoritmos que pueden alcanzar mejores cotas para el rendimiento, pero deben cumplir con ciertas restricciones como que el grafo no contenga una gran cantidad de cliques [3]. También podremos ver que para obtener representaciones simplificadas de la red que faciliten su análisis se puede aplicar algoritmos para compactar estos cliques en nodos simples que representen sus comunidades. Sin embargo, el criterio con el que se elije el manejo de cliques que comparten vértices pueden generar representaciones muy distintas entre sí. Por lo que debe tenerse cuidado en el entendimiento de lo que se busca representar al compactar un grafo.

## II. Estructuras de Datos y Algoritmos

Para la representación del grafo se consideraron listas de adyacencia almacenadas en un *unordered_map* de la biblioteca estándar de C++ (el cual implementa una tabla hash de hashing abierto) de manera que acceder a las listas por la llave del vértice sea $O(1)$. Esto también aporta la útil propiedad de que se puede acceder a las listas a través de las llaves de los nodos como indices en la tabla, simplificando de gran manera la implementación de los algoritmos.

Las listas de adyacencia están implementadas como *vectores* de la biblioteca estándar de C++, principalmente por que mantienen buena localidad espacial al momento de acceder a los vértices vecinos y por que mantienen el ingreso de nodos al final del arreglo en orden $O(1)$.

Como en muchos casos en los algoritmos utilizados es necesario trabajar con conjuntos de nodos, se hace extensivo uso de la estructura *set* de la biblioteca estándar de C++, la cual implementa un árbol binario de búsqueda. Esta estructura mantiene una colección de elementos lexicográficamente ordenados, sin repetición por lo que ayuda a simplificar el código. Sí bien, la inserción y el acceso a los elementos es de $O(\log n)$, el estándar de C++ garantiza que algunas operaciones entre contenedores se realizan en tiempo lineal para la cantidad de elementos a trabajar entre los conjuntos. En particular para las operaciones de unión, intersección y diferencia de conjuntos, que son usadas en varias ocasiones en los algoritmos implementados. De esta forma se puede reducir el costo logarítmico de usar set en varios casos.

### A. Add

Para la función *add* se ingresa cada una de las claves en la tabla hash, lo que toma $O(1)$, luego se verifica que en sus listas de adyacencia correspondiente no se encuentre la otra clave, lo que toma $O(m)$ donde m es la cantidad de elementos en la lista. Si la llave no se encuentra en la lista, esta se agrega, lo que toma $O(1)$. Finalmente la complejidad temporal es $O(m)$ con $m$ la cantidad de elementos en la lista de adyacencia correspondiente.

### B. Find

En la función *find* se utilizó un algoritmo de DFS para recorrer el grafo por lo que la complejidad temporal $O(V + E)$ donde V son los vértices del grafo y E las aristas que los conectan.

### C. Clique

Para esta función se requiere encontrar todos los cliques maximales del grafo. Estos son subgrafos completos que no pueden crecer al agregarles otro nodo adyacente. Para este trabajo se consideran como comunidades que están relacionadas entre sí, por lo que es de interés poder identificarlos. Se estudiaron dos algoritmos para encontrar todos los cliques maximales.

#### *1) Bron-Kerbosch*

Probablemente el algoritmo más reconocido para el problema de los cliques maximales [1]. Este es un algoritmo que enumera los cliques maximales mediante una estrategia de retroceso recursivo. Se consideran tres conjuntos de nodos. El conjunto $R$ que representa el conjunto de nodos que puede llegar a ser un clique maximal. El conjunto $P$ que representa los nodos que potencialmente pueden ser parte de un clique maximal y pueden ser agregados a $R$. El conjunto $X$ que

representa los nodos que ya han sido considerados por lo que no deben agregarse a R. En su forma más básica el algoritmo considera los siguientes pasos:

---

**Bron-Kerbosch**(R, P, X):
  **si** $P \cup X = \emptyset$:
    R es un clique maximal.
  **para cada** vértice *v* **en** P':
    **Bron-Kerbosch**$(R \cup \{v\}, P \cap N(v), X \cap N(v))$
    $P = P \setminus \{v\}$
    $X = X \cup \{v\}$

---

Donde P' es una copia de P y $N(v)$ son los nodos adyacentes a *v*.

Inicialmente P contiene todos los nodos del grafo y no hay nodos en X, luego para cada nodo *v* del grafo este se agrega a R. Se obtiene la intersección de P y los nodos adyacentes a *v*, ya que estos son todos potencialmente parte de un clique. Se obtiene la intersección entre X y los nodos adyacentes a *v*, ya que los nodos en X ya fueron considerados en una iteración anterior. Luego se llama recursivamente a la función sobre estos nuevos conjuntos. La función se llamara hasta que ni P, ni X contengan nodos, es decir, hasta que se encuentren todos los nodos adyacentes a *v* que formen un clique maximal. Al retornar, como *v* ya fue considerado potencialmente dentro de un clique es eliminado de P y se agrega a X. Luego se itera para el siguiente nodo en el grafo.

En el peor caso el grafo contiene el número máximo de cliques posibles en un grafo. De [4] se sabe que un grafo puede contener a lo más $3^{n/3}$ cliques, donde *n* es la cantidad de vértices del grafo, por lo que este algoritmo es óptimo en términos de cantidad de cliques y finalmente tiene un peor caso de $O(3^{n/3})$ operaciones.

*2) Algoritmo Basado en SAT*

Este algoritmo traduce el problema de encontrar los cliques maximales en un problema SAT [2]. La idea principal de este algoritmo es que puede modelarse de forma simple sobre una clase de tolerancia de un vértice *v* y una función de descernibilidad para este. La clase de tolerancia $T_x$ corresponde al conjunto de nodos adyacentes a *v* incluyéndolo. La función de discernibilidad se define como:

$$DF(x) = \wedge \left\{ \left(x_i \vee x_j\right) : x_i, x_j \in T_x \ \wedge \left(x_i \notin T_{x_j} \vee x_j \notin T_{x_i}\right) \right\}$$

Luego se define la función de discernibilidad mínima $DF'(x) = (\wedge X_i) \vee \ldots \vee (\wedge X_h)$ como la reducción a la forma normal disyuntiva de $DF(x)$.

Además se define una función mapa booleana $\partial$ como:

$$\partial_X(x) = \begin{cases} 0, & x \notin X \\ 1, & x \in X \end{cases}$$

Para decidir si un elemento *x* pertenece a un conjunto *X*.

Luego, aplicando los teoremas 3.1 y 3.2 de [2] se obtiene que los cliques maximales para un nodo *x* están dados por:

$$K_x = \{T_x \setminus X_1, \ldots, T_x \setminus X_h\}$$

Finalmente se tiene que el algoritmo para obtener todos los cliques maximales es:

---

**SAT-BasedMaximalCliquesFind**(V):
  $K = \emptyset$
  **para cada** vértice *x* **en** V:
    $K_x = \emptyset$
    **para cada** conjunto *X* **en** $DF'(x)$:
      $K_x = K_x \cup \{T_x \setminus X\}$
    $K = K \cup \{K_x\}$
  **retorna** K

---

*Donde V* es el conjunto de vértices del grafo, *K* es el conjunto de cliques maximales del grafo, $K_x$ son los cliques maximales a los que pertenece el vértice *x*, DF' es la función de discernibilidad mínima de *x*, *X* un conjunto de vértices que representa una expresión lógica en forma normal disyuntiva, y $T_x$ es la clase de tolerancia del vértice *x*.

### D. Compact

Para la función *compact* se debió considerar los casos en que los cliques maximales encontrados comparten nodos entre si. Como se está modelando una red social al compactar el grafo se debe tratar de mantener la relación entre comunidades de la red representadas por los cliques. Para esto se diseñó un algoritmo que maneja los casos antes mencionados.

Compactar cliques independientes entre sí es bastante trivial, ya que requiere de encontrar todos los nodos adyacentes al clique, luego eliminar estos nodos del grafo y reemplazar cada uno de estos en las listas de adyacencia a las que pertenecían por un nodo común enumerado como *Componente i*, donde *i* es un contador de cliques compactados. Para obtener el conjunto *Nc* de nodos adyacentes a un clique maximal *K* primero se obtiene la unión de todos los conjuntos de adyacencia de cada nodo del clique y luego se obtiene el conjunto de diferencia entre esta unión y *K* tal que $Nc = \bigcup_{v \in K} N(v) \setminus K$. El estándar de C++ garantiza que el computo de las uniones, intersecciones y diferencias toma tiempo lineal a lo más $O(N_1 + N_2 - 1)$ donde $N_1$ y $N_2$ son la cantidad de elementos en cada par de conjuntos respectivamente. En el peor caso se tiene que todas las listas de adyacencia tienen la misma cantidad de elementos por lo que el tiempo estaría acotado por $O(k \cdot (2N - 1))$, donde *k* es la cantidad de nodos en el clique. Luego deben eliminarse *k* nodos del grafo. Eliminar un nodo del grafo toma $O(e \cdot m)$ para iterar sobre la lista de adyacencia del nodo y luego para buscarlo en cada lista que contenga al nodo, donde *e* es la cantidad de nodos en la lista de adyacencia del nodo a eliminar y *m* es la cantidad de nodos en las listas de adyacencia que lo contienen. Luego se eliminan *k* nodos del grafo lo que toma $O(k \cdot e \cdot m)$. Finalmente agregar un nodo nuevo para reemplazar el clique tiene un costo de $O(1)$ por arista ingresada al grafo por lo que

tiene costo total $O(|Nc|)$. Sin embargo, como en este trabajo solo se consideran cliques maximales entre 3 y 5 elementos, el costo se reduce a tiempo constante para todas estas operaciones.

Luego para compactar cliques que comparten nodos se debe elegir una estrategia. En este trabajo se consideró comenzar a compactar todos los cliques en orden descendente de tamaño. Se saca el clique con mayor cantidad de elementos del arreglo de cliques y al resto de estos se les restan los nodos comunes con este. Luego se compacta el clique más grande y se vuelve a ordenar la lista de cliques. Este proceso se repite hasta que no quedan cliques a compactar. Los cliques que tengan menos de tres nodos no son compactados.

La idea detrás del algoritmo es que comunidades que comparten miembros seguirán estando representadas por componentes compactadas y si quedan con dos elementos o menos no es necesaria una representación compacta de estas. Esto requiere ordenar el arreglo de cliques y obtener las intersecciones entre estos por lo que a lo más tomará $O(klg\ k)$ donde $k$ es la cantidad de cliques en el arreglo. Pero como se sabe de [4] un grafo puede llegar a tener a los más $3^{n/3}$ cliques.

### E. Follow

Para la función se utilizó una cola de prioridad donde se ingresan pares de nodos con sus respectivos grados. Se utilizó una función de comparación que ordena los nodos de mayor a menor grado y en caso de que los nodos tengan el mismo grado se ordenan lexicográficamente. Luego se retornan los $n$ primeros elementos en la cola de prioridad que corresponden a los nodos con mayor grado del grafo. En caso de que $n$ sea mayor al número de nodos en el grafo se retornan todos. Por lo tanto la complejidad temporal es de $O(n \lg n)$, ya que obtener los tamaños de las listas de adyacencia es $O(1)$ y obtenerlas todas es a lo más $O(n)$. Por lo tanto domina la complejidad de la cola de prioridad.

## III. Implementación

La implementación y los comentarios correspondientes al código se entregan en el enlace del anexo y adjuntos a este informe.

## IV. Conclusiones

En el estudio de grafos, el problema de los cliques es fundamental y necesita soluciones eficientes, debido a que en el peor caso un grafo contiene una cantidad exponencial de nodos. De los algoritmos considerados para enfrentar este problema, Bron-Kerbosch sigue siendo actualmente uno de los más usados, ya que puede ser implementado con ligeras optimizaciones que aumentan su rendimiento de forma considerable, reduciendo de gran manera los retrocesos que debe realizar para continuar con la búsqueda de cliques maximales. Sin embargo como se explica en [3] hay algoritmos que pueden obtener cotas mucho más bajas para los peores casos dependiendo de las características del grafo.

Como se mencionaba anteriormente, al momento de compactar los cliques maximales es necesario tener una estrategia, ya que reducir cliques que comparten nodos puede generar confusión en la representación final. En el desarrollo de este trabajo, se probo la implementación de distintas estrategias. Varias de estas generaban grafos compactos, pero que no mantenían correctamente la relación entre los nodos originales y las comunidades de la red social. Finalmente se consideraron dos implementaciones similares, una que ordenaba los cliques de forma ascendente y otra que los ordenaba de forma descendente. Luego de ciertas pruebas se decidió por la última. Si bien ambas mantenían buenas representaciones de las comunidades, la estrategia de orden descendente logró representar mejor las relaciones para la mayoría de los casos.

## V. Anexos

Los códigos de la tarea pueden encontrarse en: https://github.com/chromosome/tarea4

## VI. Referencias


[1] C. Bron, J. Kerbosch, Algorithm 457, Finding All Cliques of an Undirected Graph, Comm. ACM 16 (1973) 575–577.

[2] Wu, H., Hao, N., & Chou, W. K. (2016). SAT-based algorithm for finding all maximal cliques. International Journal of Computational Science and Engineering, 12(2/3), 186.

[3] Tsukiyama, S., Ide, M., Ariyoshi, H., & Shirakawa, I. (1977). A New Algorithm for Generating All the Maximal Independent Sets. SIAM Journal on Computing, 6(3), 505–517.

[4] J.W. Moon, L. Moser, On cliques in Graphs, Israel J. Math. 3 (1965) 23–28.